Deep Learning for Blocking in Entity Matching: A Design Space Exploration

Saravanan Thirumuruganathan
QCRI, HBKU, Qatar
sthirumuruganathan@hbku.edu.qa

Han Li
Amazon, USA
lahl@amazon.com

Nan Tang
QCRI, HBKU, Qatar
ntang@hbku.edu.qa

Mourad Ouzzani
QCRI, HBKU, Qatar
mouzzani@hbku.edu.qa

Yash Govind
Informatica, USA
ygovind@informatica.com

Derek Paulsen
UW-Madison, Informatica, USA
dpaulsen2@wisc.edu

Glenn Fung
American Family Insurance, USA
gfung@amfam.com

AnHai Doan
UW-Madison, Informatica, USA
anhai@cs.wisc.edu

ABSTRACT

Entity matching (EM) finds data instances that refer to the same real-world entity. Most EM solutions perform blocking then matching. Many works have applied deep learning (DL) to matching, but far fewer works have applied DL to blocking. These blocking works are also limited in that they consider only a simple form of DL and some of them require labeled training data. In this paper, we develop the DeepBlocker framework that significantly advances the state of the art in applying DL to blocking for EM. We first define a large space of DL solutions for blocking, which contains solutions of varying complexity and subsumes most previous works. Next, we develop eight representative solutions in this space. These solutions do not require labeled training data and exploit recent advances in DL (e.g., sequence modeling, transformer, self supervision). We empirically determine which solutions perform best on what kind of datasets (structured, textual, or dirty). We show that the best solutions (among the above eight) outperform the best existing DL solution and the best existing non-DL solutions (including a state-of-the-art industrial non-DL solution), on dirty and textual data, and are comparable on structured data. Finally, we show that the combination of the best DL and non-DL solutions can perform even better, suggesting a new venue for research.

PVLDB Reference Format:
Saravanan Thirumuruganathan, Han Li, Nan Tang, Mourad Ouzzani, Yash Govind, Derek Paulsen, Glenn Fung, and AnHai Doan. Deep Learning for Blocking in Entity Matching:
A Design Space Exploration. PVLDB, 14(11): XXX-XXX, 2021.
doi:10.14778/3476249.3476294

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/saravanan-thirumuruganathan/DeepBlocker.

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476294

1 INTRODUCTION

Entity matching (EM) finds data instances that refer to the same real-world entity. This long-standing challenge has received much attention [16, 18, 22, 27, 64]. Most EM solutions perform blocking then matching. Given two tables A and B to match, the *blocking step* uses heuristics to quickly remove tuple pairs ($a \in A, b \in B$) judged unlikely to match. The *matching step* then applies a matcher to the remaining tuple pairs to predict match/no-match.

Recently, as deep learning (DL) became popular, many works have applied it to EM. They have shown that DL is very promising for the matching step [5, 13, 25, 46, 47, 53, 66, 80] (see [5] for a survey). In contrast, the blocking step has received far less attention. As far as we can tell, only a few recent works have applied DL to this step [4, 25, 33, 79]. A key idea is to convert each tuple $a \in A$ and $b \in B$ into an embedding vector, then quickly find tuple pairs with a high similarity score (e.g., cosine) between their vectors.

These works show the promise of DL for blocking, but are limited in important ways. First, they consider only a relatively simple way to convert each tuple into a vector, namely by combining the vectors of the words in the tuple using unweighted or weighted *averaging*. This method is called *aggregation* in the DL literature [3]. However, recent DL work for non-EM tasks (such as NLP, image processing) has developed many more ideas that can be used to combine the word vectors (e.g., sequence modeling [32], transformers [75], and self training [29, 39, 76]). Adapting these ideas for blocking in EM can potentially improve blocking accuracy.

Another limitation of the existing works is that some of them require labeled data, e.g., AutoBlock [79] uses this data to learn the weights for combining the word vectors based on semantics, position, and the surrounding context. Such labeled data is difficult to generate for blocking in many real-world EM scenarios.

In this paper we address the above two limitations and significantly advance the state of the art in DL-based blocking for EM, in the following three important ways.

Developing Self-Supervised Solutions for Blocking: We show that self-supervised techniques used in recent DL work (for non-EM tasks) can be adapted for blocking. Briefly, we define a *supervised learning task*, also called an *auxiliary task*, for which we can automatically derive labeled data from the tuples of Tables A and B. Next, we use the labeled data to train a DL model to solve the above task. Then we use *a part* of the trained DL model to produce

embedding vectors for the tuples in A and B. Finally, we quickly find tuple pairs with high similarity score between their vectors.

Defining a Rich Space of DL Solutions: Since numerous self-supervised DL solutions can be developed for blocking, we propose to organize them as well as existing solutions into a unifying space. This space provides an *important conceptual framework* that we can use to classify, compare, and understand DL solutions for blocking.

Specifically, we consider an architecture template of DL solutions that has three main steps. (1) For each tuple t in tables A and B, we compute an embedding vector for each token (i.e., word) in t. (2) We combine these vectors into an embedding vector for the entire tuple t. (3) We quickly find tuple pairs ($a \in A, b \in B$) with highly similar vectors, using a similarity measure (e.g., cosine). *For each of the above steps, we discuss multiple choices, thereby defining a rich space of DL solutions.*

In particular, for Step 2, we discuss two groups of choices: *aggregation* and *self-supervision*. Existing works use aggregation (e.g., averaging), whereas the self-supervision group contains solutions newly developed in this paper. For self-supervision, we consider four types of auxiliary tasks: *self-reproduction, cross-tuple training, triplet loss minimization*, and *hybrid*. For each type of tasks, we develop a DL solution architecture, which consists of multiple components, such as aggregator, summarizer, classifier, etc. We then show that each component can be instantiated with many possible DL models, e.g., LSTM, transformer, etc., resulting in many self-supervised solution choices for Step 2.

Evaluating Representative DL Solutions: From the large space of possible DL solutions, we select eight representative solutions: (i) SIF– a state-of-the-art aggregation solution using weighted averaging; (ii) Autoencoder, (iii) Seq2seq, (iv) Trans-encoder– three solutions that use the popular encoder/decoder framework to solve self-reproduction tasks (these solutions use feed-forward neural networks, LSTMs, and transformers as encoders/decoders, respectively); (v) CTT, (vi) CTT-cosine- two solutions that use cross-tuple training; (vii) SBERT– a solution that uses triplet loss minimization and is based on Sentence-BERT; and finally (viii) Hybrid– a solution that combines autoencoding with cross-tuple training.

We compare these solutions to one another, and to RBB, an industrial non-DL blocking solution, and BSL [51] and TB [59], two state-of-the-art non-DL solutions. Following recent EM work [13, 25, 46, 53, 62, 80], we evaluate the above solutions on multiple EM tasks using three types of datasets (structured, textual, and dirty).

Our findings are as follows. First, among the eight DL solutions described above, in terms of maximizing recall while minimizing the output size, Autoencoder performs the best on structured and dirty datasets, while Hybrid is the best on textual datasets. But even on textual datasets, Autoencoder is just slightly worse than Hybrid. This is interesting as Autoencoder's self-training method is relatively simple compared to those used by CTT and Hybrid.

Second, the best DL solutions (Autoencoder on structured/dirty and Hybrid on textual) outperform AutoBlock, the best existing DL solution for blocking (which uses labeled data), suggesting that we do not need labeled data to achieve good blocking performance. They also outperform RBB, the industrial non-DL blocking solution, on textual/dirty data, and are comparable on structured data. RBB in turn outperforms BSL and TB, the two non-DL solutions.

Finally, unioning the output of the best DL solutions (Autoencoder on structured/dirty and Hybrid on textual) with the output of the best non-DL solution (RBB) significantly increases recall, while increasing the blocking output size only modestly. This suggests that DL and non-DL solutions can be highly complementary and combining them can be a promising future research direction.

The rest of the paper is as follows. We introduce the relevant background on blocking in Section 2. We describe the components of design space in Section 3 and enumerate the concrete instantiations in Section 4. Our experimental results are presented in Section 5 followed by related work and parting thoughts in Sections 6 and 7 respectively. Additional details about the experiments can be found in a technical report [1].

2 PRELIMINARIES

Entity Matching (EM): This problem has received much attention [17, 18, 54, 71]. Many EM scenarios exist, e.g., matching two tables, matching within a table, matching a table with a knowledge base, etc. Here we will consider the following common EM scenario: given two tables A and B with the same schema, find all tuple pairs ($a \in A, b \in B$) that match, i.e., refer to the same real-world entity. Scenarios of tables with the same schema (i.e., same attributes) are very common, e.g., many real-world EM tasks match tuples *within the same table*, which can be viewed as matching two tables with the same schema. Further, even when matching tables with different schemas, most existing EM solutions [17, 18, 54, 71] examine only the attributes shared by both tables. This is because given any two tuples, these solutions must be able to *determine their similarity*, which reduces to *examining the similarities of the shared attributes*. Our solutions can still handle tables with differing schemas, but will examine only the attributes shared by the tables.

Most EM solutions perform blocking then matching. The *blocking step* uses heuristics to quickly remove pairs (a, b) judged unlikely to match. The *matching step* applies a matcher to predict match/no-match for each remaining pair. In this paper, we focus on applying DL to the blocking step.

Non-DL Work for Blocking: Blocking has received much attention (see [58, 64, 72] for surveys). Well-known blocker methods are attribute equivalence, hash, and sorted neighborhood. *Attribute equivalence (AE)* outputs a pair of tuples if they share the same values of a set of attributes. *Hash blocking* (also called *key-based blocking*) is a generalization of AE, which outputs a pair of tuples if they share the same hash value, using a pre-specified hash function. *Sorted neighborhood* outputs a pair of tuples if their hash values (also called *key values*) are within a pre-defined distance.

More complex types of blockers include similarity, rule-based, and composite blocking [17, 19, 28]. *Similarity blocking* is similar to AE, except that it accounts for dirty values, misspellings, abbreviations, and natural variations by using a predicate involving string similarity measures, such as edit distance and Jaccard [78]. *Rule-based blocking* employs multiple blocking rules, where each rule can employ multiple predicates (e.g., if the Jaccard score of the titles is below 0.6 and the years are not equivalent, then the

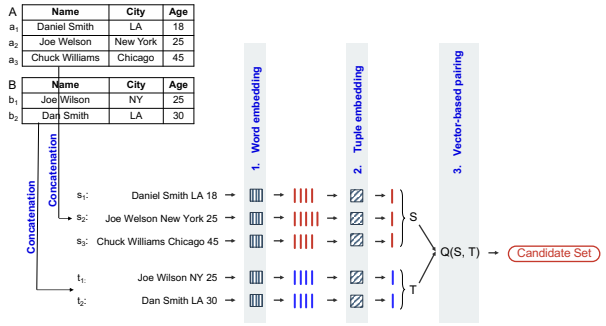

Figure 1: Our architecture template for DL-based blocking.

two papers do not match) [19, 28]. *Composite blocking* (e.g., canopy blocking [49]) generalizes rule-based blocking and can combine arbitrary blocking methods.

The above blocking methods typically assume that Tables A and B share the same schema. *Schema-agnostic blocking*, a.k.a. *token-based blocking*, is a recently developed method that does not make that assumption (it drops a tuple pairs if the tuples do not share enough tokens) [70]. Meta-blocking techniques [26, 61, 63, 69] improves upon this process by constructing a blocking graph and using it to discard redundant comparisons. Finally, *methods to learn blockers* (e.g., rule-based ones) have also been developed [19, 28, 64], typically using a set of tuple pairs labeled match/no-match.

DL Work for Blocking: Compared to the vast body of non-DL work for blocking, there have been far fewer DL works. To our knowledge, the earliest work is DeepER [25], which computes a tuple vector via unweighted aggregation of the vectors of the individual words. A recent work, AutoBlock [79], improves upon this by using a set of tuple pairs *labeled* as match/no-match to learn the weights for the aggregation. DeepBlock [33] still performs key-based blocking (a non-DL method), but optimizes it by using word embedding to compute the semantic similarity between the keys.

Our Problem Setting: We consider the problem of blocking two tables A and B with the same schema. We do not assume any labeled training data. Let C be the set of tuple pairs produced by applying a blocker to Tables A and B. We seek to develop solutions that maximizes the *recall* $|C \cap G|/|G|$, where G is the set of (unknown) true matches, while minimizing $|C|$ and the *time* taken for blocking.

3 A DESIGN SPACE OF DL SOLUTIONS

Drawing from the extensive work on blocking and DL, we now describe an architecture template for DL solutions for blocking, together with a set of choices for each module in the template.

3.1 Architecture Template & Design Space

Figure 1 shows our architecture template for DL solutions for blocking. Given two tables A and B, we first convert each tuple into a string by *concatenating* all of its attribute values. For example, tuple a_1 in Table A becomes "Daniel Smith LA 18" (see Figure 1). We use all attributes because in the absence of human input and labeled training data, we do not know which attribute is more important. So a reasonable solution is to concatenate all of them and let our proposed DL solutions learn to identify the most important features in an unsupervised way.

Table 1: The design space of DL solutions for blocking.

Template Module	Choices	
Word Embedding	Granularity (1) Word (2) Character	Training (1) Pre-trained (2) Learned
Tuple Embedding	(1) Aggregation based (a) Simple average (b) Weighted average (e.g., SIF) (2) Self-supervised *Many choices exist for each type of auxiliary tasks:* (a) Self-reproduction (b) Cross-tuple training (c) Triplet loss minimization (d) Hybrid	
Vector Pairing	(1) Hash (e.g., LSH) (2) Similarity based (a) Similarity measure: Cosine, Euclidean (b) Criteria: Threshold, KNN (3) Composite	

The resulting strings are then fed into three main modules: *Word Embedding*, *Tuple Embedding*, and *Vector Pairing*. The *Word Embedding* module converts each word in the string into a high-dimensional vector. The *Tuple Embedding* module combines these vectors into another vector representing the entire string, i.e., the original tuple. For example, recall that tuple a_1 in Table A in Figure 1 is converted into string "Daniel Smith LA 18". The Word Embedding module converts this string into four vectors (represented as four vertical red lines), then the Tuple Embedding module combines these four vectors into a single vector (see Figure 1).

At this point, each tuple in Tables A and B has been converted into an embedding vector. For example, Table A in Figure 1 has been converted into Table S of three vectors, and similarly Table B into Table T (see the right side of the figure). Finally, the *Vector Pairing* module will *quickly* search Tables S and T, using a procedure $Q(S, T)$, to find pairs of vectors that are similar. The corresponding pairs of the original tuples (from Tables A and B) are then output as the set of candidate tuple pairs for the subsequent matching step.

Our architecture template provides a set of choices for each module as shown in Table 1. Next we describe the choices, whose combination forms a design space of DL solutions for blocking.

3.2 Word Embedding Choices

This module converts each word in a string into an embedding vector. There are four main choices.

Word-level vs. Character-level Granularity: Given a sequence of words, a word-level embedding encodes each word as a fixed-dimensional vector. Typically, this is achieved through a lookup table that maps words to embeddings [52]. Any word not present in the vocabulary (e.g., rare words) triggers an out-of-vocabulary (OOV) case and is often replaced with a special token.

Character-level embedding [37] (or sub-word embeddings in general) treats each word as a sequence of sub-word units, such as individual characters, bi-grams, tri-grams, etc. and uses a neural network to produce a vector based on the character composition of the

Table 2: The many possible solutions to compute tuple embeddings.

Auxiliary Tasks	Solution Architectures	Instantiation Examples
Self-Reproduction	Aggregator + Encoder + Decoder	• Aggregator: SIF, LSTM, . . . • Encoder/Decoder: Feed-Forward NN, LSTM, Transformer
Cross-Tuple Training	Aggregator + Summarizer + Classifier	• Aggregator: SIF, LSTM, . . . • Classifier: Feed-Forward NN, LSTM, Cosine
Triplet Loss Minimization	Aggregator + Loss Minimizer	• Aggregator: BERT, . . .
Hybrid	Many possible architectures exist, e.g.: • encoder + summarizer + classifier • SBERT aggregator + summarizer + classifier	• Each architecture has many components • Each component has many instantiation choices as listed above

word. This approach can transparently handle the morphological aspects of words (such as 'data', 'database' and 'dataset'), can produce an embedding for any out-of-vocabulary words, and is robust to common misspellings. Thus, this may work well for EM scenarios with custom vocabularies and widespread misspellings [53].

Pretrained vs. Learned Embeddings: An orthogonal design choice relates to how the embeddings were trained. Popular word-level embeddings such as word2vec [52] and GloVe [67] and character-level embeddings such as fastText [9] often come with pre-trained embeddings that are trained on a large generic external corpus such as Wikipedia, Common Crawl or PubMed. Alternatively, they could be trained on the dataset for which EM has to be performed.

3.3 Tuple Embedding Choices

This module combines the embeddings of the words in a tuple into an embedding for the entire tuple. A key challenge is to ensure that similar tuples have similar embeddings without labeled data. To address this, we consider two broad categories of promising DL techniques: aggregation and self-supervision.

Aggregation Methods: These methods apply an aggregation function $F : \mathbb{R}^{d_e \times \cdot} \rightarrow \mathbb{R}^{d_u}$ to produce a tuple embedding $\mathbf{u}_t$. The most popular method is averaging, e.g., DeepER [25] uses unweighted averaging where each word embedding has equal weight, while SIF [3] uses weighted averaging.

Since later we experiment with SIF, we now describe it in more details. SIF works as follows. First, for each tuple, we calculate a weighted average over the word embeddings to obtain the aggregation vector. Given a word w in the tuple string, the weights of its word embedding is given as $f(w) = a/(a + p(w))$ where a is a hyper-parameter and $p(w)$ the normalized unigram frequency of w in the dataset. Next, we calculate the first principal component of the aggregation vectors using PCA. Finally, we calculate the tuple embedding for each tuple by subtracting the projection of its aggregation vector over the first principal component. Specifically, let $\mathbf{v}_t$ be the aggregation vector for the tuple t, and $\mathbf{p}$ be the first principal component, the tuple embedding $\mathbf{u}_t = \mathbf{v}_t - \mathbf{p}\mathbf{p}^T\mathbf{v}_t$. SIF is shown to perform comparably to complex models for NLP tasks such as text similarity [3] and generalizes the unweighted averaging approach of DeepER [25].

As described, aggregation methods do not involve any learning and can be implemented efficiently. But they take a bag-of-words approach where the ordering information is disregarded. Thus, the tuples 'A bought B' and 'B bought A' will have the same embedding. Further, they cannot handle polysemy where the same word/phrase could have multiple meanings, thus producing the same word embedding for 'Apple' in the phrases 'Apple tv' and 'Apple tree'. More sophisticated embedding approaches, which we discuss next, address these limitations.

Self-Supervised Methods: These methods adapt the self-supervision idea popular in recent DL works [21, 23, 34, 52, 56]. They work as follows: (1) Define a *supervised learning task*, also called an *auxiliary task*, for which we can *automatically* derive *labeled training data* from the tuples of Tables A and B. (2) Solve the above task using a *DL model* trained on the labeled data. (3) Use parts of the trained DL model to produce embeddings for the tuples in Tables A and B.

In what follows, we consider four types of auxiliary tasks (see Table 2): self-reproduction, cross-tuple training, triplet loss minimization, and hybrid. For each type, we discuss a promising DL solution architecture. We show that such an architecture typically consists of multiple components (e.g., aggregator, summarizer, etc.), and that for each component, we can have many possible instantiations (e.g., an aggregator can be a simple unweighted averaging, a feed-forward NN, or a LSTM), as illustrated in Table 2. Thus, *each combination of the instantiations produce a solution for tuple embedding, giving rise to a rich set of such solutions.*

It is important to note that, as far as we know, existing work (e.g., DeepER and AutoBlock) has considered only aggregration methods for blocking (see Section 2). Thus *a key technical novelty of this work is that we adapt the idea of self supervision popular in the recent DL literature to blocking for EM, and that we develop a range of such solutions*, which we describe in the next four subsections.

3.4 Self-Reproduction Methods

These methods use the simplest auxiliary task: *self-reproduction*. Roughly speaking, they take a tuple t, feed it into a neural network (NN) to output a *compact* embedding vector $\mathbf{u}_t$, such that if we feed $\mathbf{u}_t$ into a second NN, we can recover the original tuple t (or a good approximation of t). If this happens, $\mathbf{u}_t$ can be viewed as a good *compact summary* of tuple t, and can be used as the tuple embedding of t. The above two NNs are called *encoder* and *decoder*, respectively. Such so-called *autoencoder frameworks* have been used extensively in other tasks, such as dimensionality reduction. But to our best knowledge they have not been used for blocking for EM.

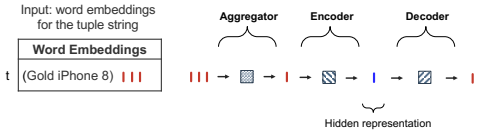

Figure 2: The autoencoder model architecture.

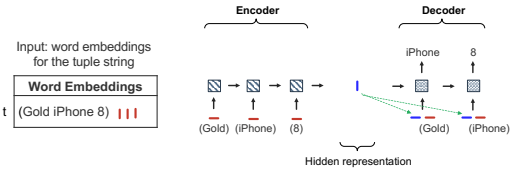

Figure 3: The seq2seq model architecture.

We now describe two autoencoding methods, Autoencoder and Seq2seq, which we later evaluate in depth. Then we build on these two to discuss other possible autoencoding methods.

Autoencoder - A Feed-Forward Neural Network Method: This is a relatively simple autoencoding method (hence the generic name Autoencoder). Recall that we want to build a model that accepts the word embedding sequence $\mathbf{e}_t$ (of a tuple t) as input, and generates an output vector $\mathbf{o}_t$ that recovers the information in $\mathbf{e}_t$. Ideally, we want the model to reproduce $\mathbf{e}_t$ exactly (i.e. $\mathbf{o}_t = \mathbf{e}_t$), and train the model for this goal.

As shown in Figure 2, our model consists of an aggregator, an encoder and a decoder. We use a two-layer feed-forward NNs with the Tanh activation function for both the encoder and decoder. The feed-forward NN cannot accept word embedding sequences of variable lengths as input. Hence, we modify the classic autoencoder model by performing an aggregation operation $f(\cdot)$ in the first step to convert $\mathbf{e}_t \in \mathbb{R}^{d_e \times \cdot}$ into a fixed-size vector $\mathbf{v}_t = f(\mathbf{e}_t)$ where $\mathbf{v}_t \in R^{d_e}$. For the aggregation implementation, we use the SIF model [3] which calculates a weighted average $\mathbf{v}_t$ over the word embeddings in $\mathbf{e}_t$. Next, the encoder receives $\mathbf{v}_t$ as the input and generates a hidden vector $\mathbf{u}_t \in \mathbb{R}^{d_u}$. Then the decoder uses $\mathbf{u}_t$ to produce the output $\mathbf{o}_t \in \mathbb{R}^{d_e}$ such that it approximates $\mathbf{v}_t$.

The training loss on the tuple t is defined as $l_t = ||\mathbf{v}_t - \mathbf{o}_t||_2^2$, which is the squared ℓ_2 distance between the aggregation vector and the output vector. The training goal is to update the parameters in the encoder and the decoder such that the training loss is minimized. Once the training is done, *to generate the tuple embedding for a given tuple t, we feed the word embedding sequence $\mathbf{e}_t$ of t to the aggregator followed by the encoder. We use the generated hidden representation vector $\mathbf{u}_t$ as the tuple embedding vector for t.*

Seq2seq - A Sequence-to-Sequence Method: Autoencoder is not sequence aware. It produces the same aggregation vector for any permutation of the input string. We now describe Seq2seq, an approach that is sequence aware: given a word sequence w_t and the corresponding embedding sequence $\mathbf{e}_t$, it reproduces w_t.

Similar to Autoencoder, Seq2seq also consists of an encoder and a decoder. But they are LSTM-RNNs [32], which can handle sequences with variable length, as shown in Figure 3. The LSTM-based encoder takes each embedding vector in $\mathbf{e}_t$ one-by-one to get a hidden representation vector $\mathbf{u}_t$. Formally, given the i-th embedding $\mathbf{e}_t[i]$ in $\mathbf{e}_t$, the LSTM unit $LSTM_{enc}$ takes $\mathbf{e}_t[i]$ and the hidden state $h_{enc}[i-1]$ from the previous step as input to produce a new hidden state $h_{enc}[i] = LSTM_{enc}(\mathbf{e}_t[i], h_{enc}[i-1])$. We denote the last hidden state output as $\mathbf{u}_t$, which will be used as the context for the decoder to recover the word sequence. For the first hidden state input $h_{enc}[0]$, we randomly initialize the vector.

For the decoder, we use a combination of an LSTM $LSTM_{dec}$ and a one-layer feed-forward NN *Out* to recover the word sequence w_t. First, the hidden state $\mathbf{u}_t$ will be used as the initial state $h_{dec}[0]$ for $LSTM_{dec}$, which serves as the context for the sequence decoding. Second, we pass the i-th word embedding $\mathbf{e}_t[i]$, the context $\mathbf{u}_t$, and the previous hidden state $h_{dec}[i-1]$ to the decoder and try to predict the next word in the sequence $w_t[i+1]$.

This can be further decomposed into two sub-steps. First, we pass the three vectors $\mathbf{e}_t[i]$, $\mathbf{u}_t$, and $h_{dec}[i-1]$ to the LSTM decoder to get the hidden state $h_{dec}[i] = LSTM_{dec}(\mathbf{e}_t[i], \mathbf{u}_t, h_{dec}[i-1])$. Once we have the hidden state $h_{dec}[i]$, we send it through the feed-forward NN *Out*, to output a score vector $\mathbf{o}_i = Out(h_{dec}[i]) \in \mathbb{R}^{|V|}$ over the entire word vocabulary V . The j-th index $\mathbf{o}_i[j]$ is a score indicating how likely the next word in the sequence will be the j-th word in the vocabulary V . As we want to predict the next word $w_t[i+1]$ in the sequence, suppose $w_t[i+1]$ is the k-th word in the vocabulary, we want $o_i[k]$ to be the largest value in $\mathbf{o}_i$.

For the model training objective, we use the cross-entropy loss for each position i to maximize the value in o_i whose index corresponds to the index of the word $w_t[i+1]$ in the vocabulary. Note that in the decoding procedure, besides using $\mathbf{u}_t$ as the context to the LSTM, we also use it as a part of the input for each decoding step, as shown by the blue bar in Figure 3. The reason is that we want to learn $\mathbf{u}_t$ such that it will affect the recovery of each word in the sequence. This will reduce the importance of the LSTM in decoding and force $\mathbf{u}_t$ to summarize the information in $\mathbf{e}_t$ well to be able to recover the word sequence. *Given the embedding sequence $\mathbf{e}_t$ for a tuple t, we apply $\mathbf{e}_t$ to the LSTM encoder $LSTM_{enc}$, and the last hidden state output $\mathbf{u}_t$ will be used as the tuple embedding.*

More Self-Reproduction Methods: As described, a promising DL solution architecture to solve the self-reproduction task consists of three components: aggregator (optional), encoder, and decoder. So far, we have discussed one instantiation for the aggregator (SIF) and two possible instantiations for the encoder and decoder (feed-forward NN and LSTM). However, we can supply more instantiations for these components. For example, one could replace LSTM with standard transformer (encoder/decoder). We refer to this instantiation as Trans-encoder. Alternatively, one could use LSTM (instead of SIF) for the aggregator in Autoencoder making the latter sequence-aware. Each combination of these instantiations produce a possible method to compute tuple embeddings.

3.5 Cross-Tuple Training Methods

The self-reproduction approach exploits information within a single tuple to generate tuple embeddings. Our next approach, CTT, exploits information across tuples. The key idea is to perturb the tuples of Tables A and B to generate synthetic labeled data, which is

a set of tuple pairs (t_i, t_j) with match/no-match labels, then use this data to train a DL model to produce tuple embeddings, such that the embeddings of a matching tuple pair are closer to each other while those of non-matching tuple pairs are farther from each other. To explain CTT, we first consider an ideal scenario where labeled data is available, then show how to generate synthetic labeled data.

Ideal Model Implementation: Suppose we are given a set of tuple pairs $C \subseteq A \times B$ with match/non-match labels. We can then train a classifier to predict for each pair in C the correct label. This is achieved in two steps. First, given a tuple pair $(a \in A, b \in B)$ we transform their embedding sequences $\mathbf{e}_a$ and $\mathbf{e}_b$ into tuple embeddings $\mathbf{u}_a$ and $\mathbf{u}_b$. Next, we train a classifier that takes $\mathbf{u}_a$ and $\mathbf{u}_b$ and predicts the correct label. The key insight is that in order to do well on the prediction task, the model has to learn to generate effective tuple embeddings that is relevant for correctly matching predictions from tuple comparisons.

Ideally we generate the set C as follows. Suppose that we have the set $M \subseteq A \times B$ containing all matching pairs. We can simply take all pairs in M as the positive training instances. To generate the negative training data, we can select a set of tuple pairs from $A \times B$ that are not in M as the negative training instances. Specifically, let $E = A \cup B$. For each tuple $t \in E$, we randomly select a set $S_t \subseteq E$ of p tuples (where p is a hyperparameter), to form a set of non-matching pairs $N_t = \{(t, s) | s \in S_t\}$ satisfying each pair $(t, s) \notin M$. We repeat this procedure for every tuple $t \in E$, and finally we take the union $N = \cup_{\{t \in E\}} N_t$ as the negative training data.

We then use the labeled dataset $C = M \cup N$ to learn the embeddings. Figure 4 shows the model architecture. This model consists of three modules: an aggregator, a Siamese summarizer, and a classifier. Given a pair of word embedding sequences e_1 and e_2 from C, we first apply the aggregator to convert each embedding sequence into a fixed-size vector respectively, denoted as $\mathbf{v}_1 \in \mathbb{R}^d_e$ and $\mathbf{v}_2 \in \mathbb{R}^d_e$. We use the SIF model for the embedding sequence aggregation.

Next, for each of v_1 and v_2, we use a Siamese summarizer, which is a two-layer feed-forward neural network, to generate a summarized vector $\mathbf{u}_1 \in \mathbb{R}^{d_u}$ and $\mathbf{u}_2 \in \mathbb{R}^{d_u}$, respectively. Then, we take an element-wise absolute difference between $\mathbf{u}_1$ and $\mathbf{u}_2$, and send it to the classifier, which is a two-layer feed-forward neural network, to predict a label indicating the input pair a match or a non-match.

The training goal is to learn the model parameters in both Siamese summarizers such that the predictions are the same as the gold labels in the training data. The use of a Siamese network reduces the capacity of the model as both summarizers use the same model parameters. *Given the embedding sequence* $\mathbf{e}_t$ *for tuple* t *in* A *or* B*, we apply the aggregator followed by the Siamese summarizer in the trained* CTT *model. The generated summarization vector* $\mathbf{u}_t$ *is used as the tuple embedding.*

Approximating the Ideal Training Data. In order to implement the ideal approach mentioned above, we need to know all matching pairs in $M \subseteq A \times B$ in advance. However, this means that we have already solved the EM problem! We next propose a data generation procedure for approximating the ideal training data without having access to M. Once again, let $E = A \cup B$. For each tuple $t \in E$, we have to generate one positive training instance and p negative instances.

We use a simple but effective heuristic for generating positive tuple pairs. Given a tuple t, we obtain the word sequence w_t through

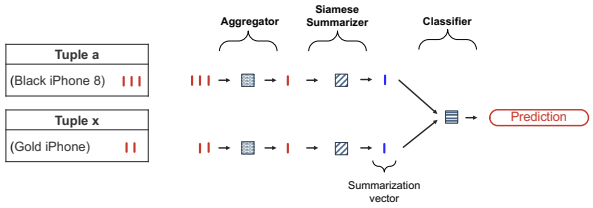

Figure 4: The CTT model architecture.

concatenation. We need to generate another tuple t' (ideally in E) that is likely to be a match for t. However, at this point we do not know which tuple matches t. To solve this conundrum, we *generate* a *synthetic* matching tuple string by randomly selecting a subset of words from w_t, denoted as w'_t. Since w'_t is selected from w_t, we associate a label "1" for this pair (w_t, w'_t) indicating it is a match. Varying the proportion of overlapping strings increases the likelihood that it is indeed a match. In our experiments, we ensure that the synthetic tuple has at least 60% overlap.

To generate a negative instance, we randomly select a tuple s in E (excluding t), concatenate the attribute values of s to get w_s, and use w_s as the second tuple. As the tuple s is randomly selected, it will very likely be a non-match to t since usually matches are rare compared to non-matches. We associate a label "0" indicating a non-match for the pair (w_t, w_s). We repeat this procedure p times with p random tuple selections in E, which gives us p negative training instances. We repeat the procedure for every tuple in the tables A and B, and finally take the union of the training instances for every tuple as the approximated training data. Once we have the approximated training data, we train a CTT similar to the way we train in the ideal training scenario.

A Range of CTT Methods: As described, the CTT approach uses a DL architecture that consists of three main components: aggregator, summarizer, and classifier. So far, we have used SIF, Siamese summarizer, and a two-layer feed-forward NN for these components, respectively. However, we can easily provide more instantiations such as replacing SIF with LSTM in the aggregator. Each combination of these instantiations (see Table 2) produce a possible method to compute tuple embeddings.

An interesting extension of CTT that we consider is CTT-cosine. In this solution, we replace the feed-forward neural network used in the CTT classifier with a simple cosine similarity. This is appropriate as a number of options in the vector pairing module are based on cosine similarity. Given a tuple t, a positive perturbation t' and p tuples from the negative set N_t, CTT-cosine seeks to fine-tune the embeddings such that the cosine similarity score between t and t' is maximized.

3.6 Triplet Loss Minimization Methods

This approach adapts the triplet method used in many recent DL tasks [7, 15, 43]. It works as follows. First, we generate synthetic training data, which is a set of triplets. Each triplet (x, y, z) consists of a tuple x, a *synthetic* tuple y that matches x, and a tuple z that does not match x. For each tuple t_i in Tables A or B, we generated L perturbations $p_{i,1}, \ldots, p_{i,L}$ by randomly removing up to 40% of words

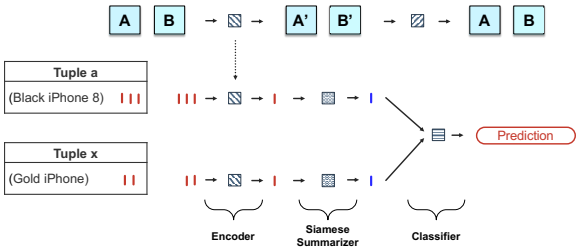

Figure 5: The Hybrid **model architecture.**

from t_i. Then we pick L random tuples $n_{i,1}, \ldots, n_{i,L}$. Finally, we generate L triplets of the form $\{(t_i, p_{i,1}, n_{i,1}), \ldots, (t_i, p_{i,L}, n_{i,L})\}$.

Next, we train a DL model such that the model produces vector embeddings for x, y, and z, and the embeddings for x and y are close while those for x and z are far apart. We use a pretrained BERT [21] to produce embeddings and the SBERT (Sentence-BERT) [68] approach for tuple aggregation. We use the Triplet loss function [7, 15, 43] defined as

$$\max\left(||Emb(x) - Emb(y)||^2 - ||Emb(x) - Emb(z)||^2 + \alpha, 0\right)$$

Emb(x) provides the embedding for tuple x and α is the distance margin to be ensured between positive and negative tuple pairs.

BERT is based on transformers, a model more powerful than LSTMs and commonly used to capture sequential information. BERT has been used to summarize a tuple for the matching step of EM [13, 46]. We believe it can also be used to summarize a tuple, i.e., produce a tuple embedding, for blocking. Such embeddings can capture not just the word ordering but also other contextual information, allowing us to distinguish for instance the word "Apple" in "Apple tv" vs "Apple tree".

After training, *we use the aggregator component of* SBERT *to produce vector embeddings for the tuples.* Further, we can replace the aggregators in the SBERT solution with any NN that can aggregate a sequence of vector embeddings into a single vector embedding, to obtain another method to compute tuple embeddings.

3.7 Hybrid Methods

Each method to compute tuple embeddings that we have discussed so far uses just *one* auxiliary task: self-reproduction, cross-tuple training, or triplet loss minimization. We can develop many "hybrid" methods, each of which uses *two or more* auxiliary tasks.

For example, we can combine Autoencoder and CTT to generate tuple embeddings that take into account both in-tuple and cross-tuple information. To do so, we use a stacked training procedure consisting of two sub-training tasks. Given the word embedding sequences of tuples in A and B, we train an Autoencoder model M_1 as described earlier, and then a CTT model M_2. The training step for M_2 is as described earlier, except that we use a modified version of the original CTT model. Instead of SIF, we use the (trained) encoder of M_1 as the aggregator for M_2.

Note that M_1 and M_2 are stacked by training M_1 first then M_2, as opposed to training M_1 and M_2 jointly. The reason is that we want to keep the two models M_1 and M_2 separate to avoid cross-tuple information diffusing to the model parameters of M_1 (if we do the joint training), such that M_1 does not summarize the in-tuple information well. Once the training is completed, this stacked model could be used to generate tuple embeddings. *Given the embedding sequence* $\mathbf{e}_t$ *for tuple* t *in* A *or* B*, we feed* $\mathbf{e}_t$ *to the aggregator (which is the encoder in* M_1*) followed by the Siamese summarizer in* M_2*, and use the output as the tuple embedding vector.*

Figure 5 illustrates the proposed architecture, which we will simply call Hybrid. Note that many other possible "hybrid" solutions exist, e.g., using a BERT module (trained in the SBERT solution) as the aggregator, see Table 2. While such hybrid models are quite powerful in terms of the model capacity and expressiveness power, they also take longer time to train.

3.8 Vector Pairing Choices

So far we have discussed many methods to compute tuple embeddings. We now discuss vector pairing. Let S and T be the tables of tuple embedding vectors corresponding to Tables A and B, respectively. We want to *quickly* search S and T, using a procedure $Q(S, T)$, to find pairs of similar vectors. To do so, we adapt the major non-DL blocking approaches, which can be categorized as hash-, sorting-, similarity-based, composite (e.g., canopy clustering, rule-based), and schema agnostic. Sort-based and schema agnostic approaches rely on the symbolic content of the tuples. These approaches do not work well for embeddings which are multi-dimensional real-valued vectors that do not have a meaningful lexicographic order for sorting or measuring overlap. Thus, we are left with only hash-based, similarity-based, and composite approaches.

Hashing-based Pairing: This approach hashes each tuple represented by an embedding vector and keeps only tuple pairs that share the same hash value. It can be implemented quite efficiently. Since the tuple embeddings are numeric vectors, Locality Sensitive Hashing (LSH) [44] is a good choice which hashes similar items into the same bucket with high probability. Both DeepER [25] and AutoBlock [79] use hashing-based pairing.

Similarity-based Pairing: This approach keeps only those tuple pairs where the tuples are quite similar, based on a similarity measure such as cosine similarity and Euclidean distance. A natural choice is to keep only tuple pairs where the similarity score exceeds a threshold (e.g., 0.7). Another choice is to keep k nearest neighbors (kNN). For example, say we use cosine measure, then for each tuple $a_i \in A$ with the tuple embedding $\mathbf{u}_{a_i}$, we first calculate the cosine scores between $\mathbf{u}_{a_i}$ and every $u \in T$. Then, we pick k tuples $B' \subseteq B$ whose corresponding embeddings vectors have the highest k cosine similarity scores. Those k pairs (a_i, b_j) where $b_j \in B'$ will be included in the candidate set.

Composite Pairing: This approach combines the above approaches. For example, we may first apply LSH to get a set of hash buckets, then selecting k pairs with the highest cosine similarity scores in each bucket.

4 REPRESENTATIVE DL SOLUTIONS

The previous section describes a space of DL solutions for blocking. Given the large number of these solutions, in this paper we will evaluate in depth only *eight representative solutions*, which correspond to DL models of varying complexity. The eight solutions

differ significantly in their choices for the tuple embedding module, and are named accordingly.

Each of our solutions (except SBERT and Trans-encoder) use fastText [9], a pre-trained character-level embedding, for the word embedding module. fastText can handle word morphologies and out-of-vocabulary words, and is robust to common misspellings. As such, it is a good choice for the word embedding module, and has also been used by multiple recent work on DL for EM [25, 53, 79]. Our transformer based solutions, SBERT and Trans-encoder, use BPE (Byte Pair Encoding), which is also based on sub-word units such as individual characters. Furthermore, all eight solutions use top-k cosine similarity for the vector pairing module. This allows us to cleanly control the desired size of the blocking output (via the parameter k), a highly desirable property for practical applications. Table 3 gives a summary of how each component was instantiated. In the next section we evaluate the above eight solutions in depth.

Table 3: Summary of our proposed solutions.

Solution	Instantiation Details
SIF	Aggregator: SIF
Seq2seq	Encoder/Decoder: LSTM
Autoencoder	Aggregator: SIF; Encoder/Decoder: Feed-Forward network
SBERT	Aggregator: BERT
CTT	Aggregator: SIF; Summarizer: Siamese network; Classifier: Feed-Forward network
Hybrid	Autoencoder + CTT
Trans-encoder	Encoder/Decoder: Transformer
CTT-cosine	Aggregator: SIF; Summarizer: Siamese network; Classifier: Cosine

5 EMPIRICAL VALUATION

Datasets: We use datasets from a diverse array of domains and different sizes (see Table 4). All of these datasets except Hospital, which is a private dataset, are publicly available and have been used in previous work on EM. For structured EM, we use 6 datasets. For textual EM, we use 3 datasets with 2-3 attributes that are textual blobs (e.g., title, description). For dirty EM, we focus on one type of dirtiness, which is widespread in practice [53], mainly due to information extraction glitches, where attribute values are 'injected' into other attributes (e.g., the value of 'brand' is missing and appears in attribute 'title'). Textual and dirty datasets are derived from the corresponding structured datasets. We also conducted additional experiments on three real-world datasets – Restaurants [24], Books [24] and Cora [50] that contain real noise such as misspellings, missing values and incorrect entries.

Methods: We evaluate the eight developed DL solutions, as well as state-of-the-art existing DL and non-DL solutions. Let C be the candidate set, i.e., the output of blocking on two tables A and B, and let G be the set of true matches between A and B (note that we know G for all datasets that we experimented with). Then recall is measured as $|G \cap C|/|G|$, and candidate set size ratio (CSSR) is measured as $|C|/|A \times B|$. Ideally, we want high recall, low CSSR and low runtime.

Table 4: Datasets for our experiments.

Type	Dataset	Table A	Table B	#Matches	#Attr
Structured	Amazon-Google$_1$	1,363	3,226	1,300	4
	Walmart-Amazon$_1$	2,554	22,074	1,154	6
	DBLP-Google$_1$	2,616	64,263	5,347	4
	DBLP-ACM$_1$	2,616	2,294	2,224	4
	Hospital$_1$	1,786	1,786	3,949	7
	Songs-Songs$_1$	1,000,000	1,000,000	1,292,023	5
Textual	Amazon-Google$_2$	1363	3,226	1,300	2
	Walmart-Amazon$_2$	2,554	22,074	1,154	2
	Abt-Buy	1,081	1,092	1,097	3
Dirty	DBLP-ACM$_2$	2,616	2,294	2,224	4
	Hospital$_2$	1,786	1,786	3,949	7
	Songs-Songs$_2$	1,000,000	1,000,000	1,292,023	5

We implemented our solutions in PyTorch, and used a server with one Intel Xeon E5-2686 CPU, 64 GB RAM, and one NVIDIA V100 GPU for experiments. We train the DL models with mini-batch gradient descent, and use Adam [38] as the optimization algorithm. Recall that we use fastText for attribute embedding (except SBERT and Trans-encoder where we use BPE) and top-K cosine for vector pairing. We used the well-known FAISS [35] library for top-k cosine search due to its GPU optimizations.

5.1 Recall and Candidate Set Size

We begin by evaluating the eight DL solutions in terms of recall and candidate set size. To do so, we plot the R-C (recall-candidate set size ratio) curve for each solution, to show how these quantities change as we vary the top-K value in the vector pairing module (e.g., $K = 10$ means that each tuple in Table A is paired with 10 tuples in Table B with the highest cosine scores).

Structured Data: Figure 6 compares the R-C curves on six structured datasets. The x-axis shows the recall while the y-axis shows the candidate set size ratio (CSSR) for all datasets except Song-Song. The large size of Song-Song (1M tuples) produces very small values for CSSR (e.g., for $K = 100$, the candidate set size is 100M, producing $CSSR = 0.0001$, which is too small to show on the plot). So we report the value of K instead on the y-axis. As K increases, both recall and candidate set size ratio increase. An R-C curve closer to the bottom-right corner of the plot indicates a better solution, as this corresponds to a smaller candidate set and a higher recall.

The plots show that all *eight* solutions can achieve high recall (above 90%) with relatively small candidate sets. Autoencoder achieves the best performance overall, as its curves are consistently closest to the bottom-right corner. Surprisingly Hybrid is the best only for Walmart-Amazon$_1$, despite having the best representation power among the eight solutions. Our ablation analysis shows that this is due to the quality of the approximated training data that it uses. Seq2seq is significantly outperformed by other solutions, including SIF for structured and dirty datasets. This is due to tuples in structured and dirty datasets being relatively short (4-7 attributes). The number of distinct attribute values in these tables were limited. Furthermore, the sequential information in structured datasets (such as manufacturer and price) were non-existent unlike textual datasets In contrast, the vocabulary size for textual datasets is much higher where Seq2seq performs much better.

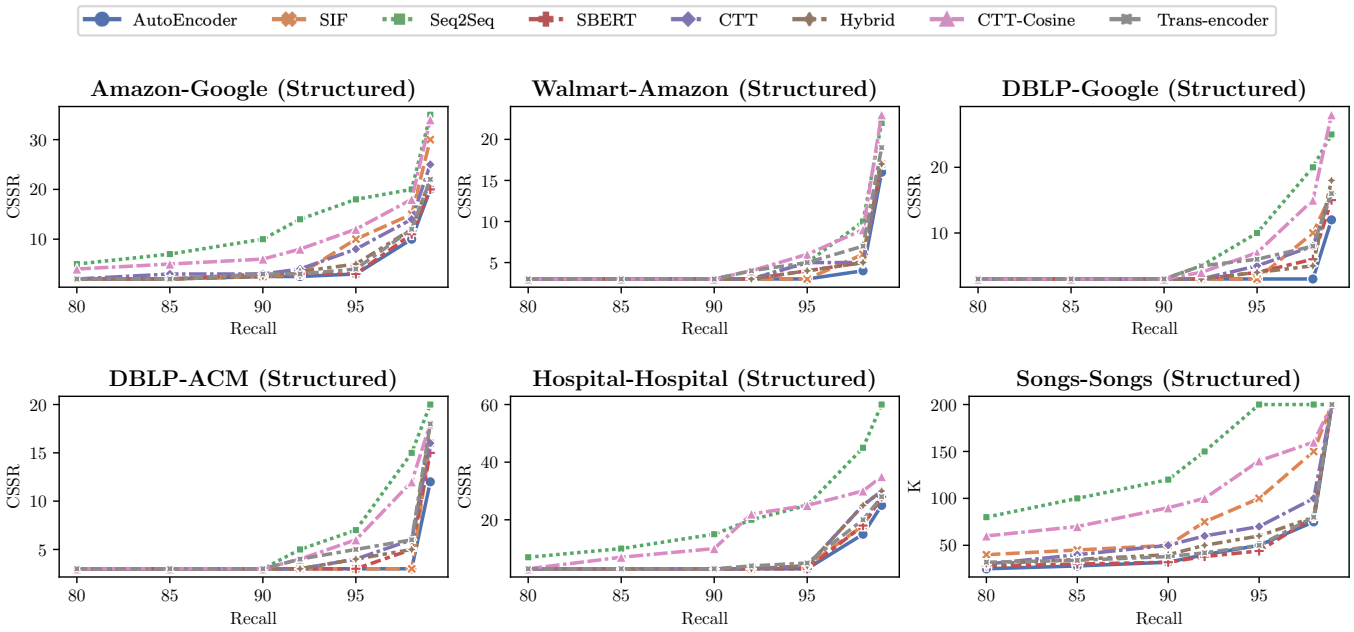

Figure 6: R-C curve comparison of DL solutions for structured datasets.

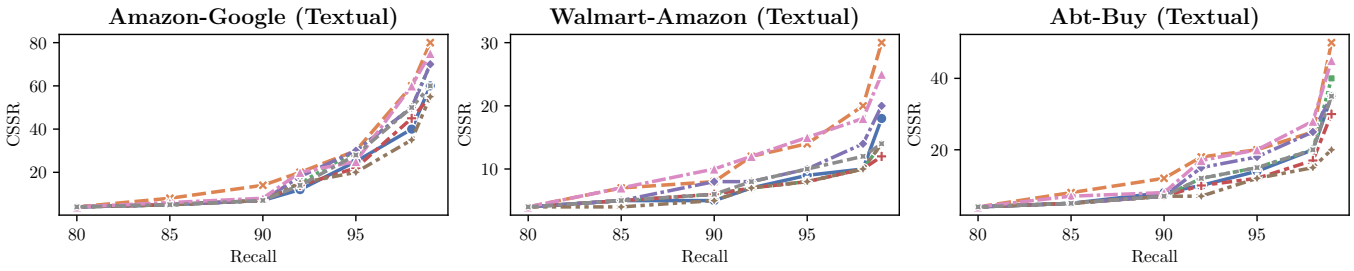

Figure 7: R-C curve comparison of DL solutions for textual datasets.

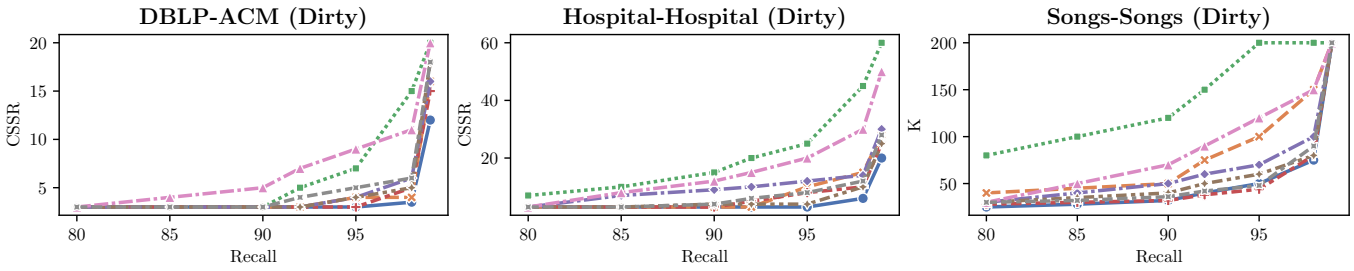

Figure 8: R-C curve comparison of DL solutions for synthetically created dirty datasets (same legend as that of Figure 6).

Textual Data: Figure 7 shows the R-C curves on textual datasets. The performance of the solutions are quite similar, with the R-C curves close to each other. These R-C curves are further from the bottom-right corner of each plot than those of structured datasets, This implies that for datasets with long textual attributes, it is much more challenging for DL methods to extract useful information from each tuple and generate good blocking results. However, as we shall show later, DL based methods still outperform non-DL methods for this scenario. The plots show that Hybrid achieves the best performance on average, suggesting that for textual data, capturing

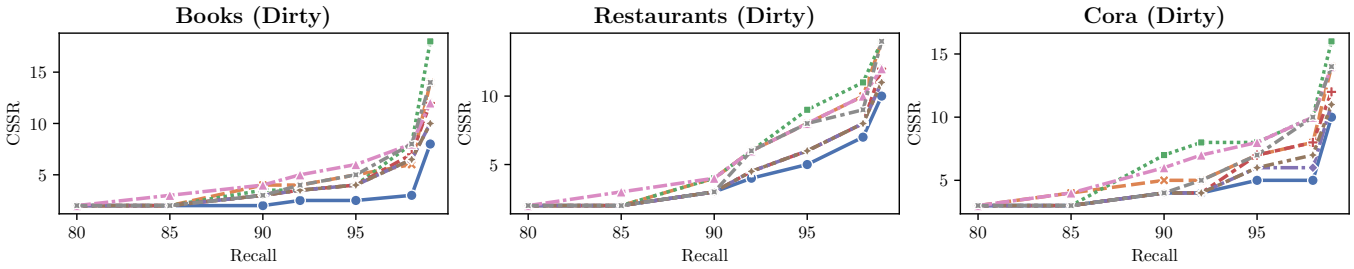

Figure 9: R-C curve comparison of DL solutions for real-world dirty datasets. Legend is the same as that of Figure 6

cross-tuple information helps generate better tuple embeddings. SBERT also performs quite well. It also uses cross-tuple information and thus further supports using this information for textual data. Interestingly, Autoencoder is a close third despite using only the in-tuple information.

Dirty Data: We conducted experiments on two different variants of dirty datasets. Figure 8 shows the R-C curves on dirty datasets that were obtained by synthetically corrupting the structured datasets. Figure 9 shows the results for three datasets, namely Restaurants, Books and Cora, with real-world noise. Overall, they are quite similar to the curves for structured datasets, with Autoencoder being the best. This is not entirely surprising, because dirtiness in these datasets means moving some attribute values around, which can degrade solutions that exploit sequential information (e.g., Seq2seq), but has no effect on those that do not (e.g., Autoencoder).

5.2 Runtime

Next we evaluate our solutions in term of runtime. We focus on the major "time sinks": the training time for the tuple embedding module and the time of the vector pairing module.

Training Time for the Tuple Embedding Module: Table 5 shows the training time for each dataset. There is no learning involved for SIF while SBERT uses a pre-trained model. The training time of Seq2seq is more than an order of magnitude higher than the time of others. Unfortunately, LSTMs cannot be easily parallelized due to its sequential nature. The other solutions are quite efficient and can scale well even for large datasets (e.g., Song-Song with 1M tuples). Autoencoder which is the best for structured and dirty data is especially efficient to train.

Runtime of the Vector Pairing Module: We found that the FAISS [35]-based vector pairing module, that uses GPU acceleration for fast similarity search required less than a minute to generate candidate set for $K = 100$ for all datasets except Song-Song, for which it required slightly less than 35 minutes.

Evaluating Different Vector Pairing Choices: We have also performed extensive micro-benchmarks to evaluate vector pairing choices. We found that in all EM problem types, top-K cosine outperforms threshold-based cosine, and is much better than LSH. We have provided additional details in the tech report [1].

Table 5: Training time for the tuple embedding module.

Dataset	Auto Encoder	Seq2 Seq	CTT	Hybrid
Amazon-Google$_1$	1.4m	14.1m	7.3m	8.9m
Walmart-Amazon$_1$	2.4m	50.2m	10.6m	13.7m
DBLP-Google$_1$	4.4m	7.7h	31m	42m
DBLP-ACM$_1$	1.6m	38.1m	8.3m	10.1m
Hospital$_1$	1.1m	9.7m	5.8m	6.3m
Song-Song$_1$	22m	28h	1.8h	3.1h
Amazon-Google$_2$	4.1m	32m	16.2m	20.8m
Walmart-Amazon$_2$	6.4m	1.9h	56m	1.3h
Abt-Buy	3.5m	18.6m	12m	18m
DBLP-ACM$_2$	1.8m	40.6m	8.8m	12.1m
Hospital$_2$	1.4m	10.4m	7.2m	7.5m
Song-Song$_2$	23m	29h	2.1h	2.9h

5.3 Comparing with Existing DL Solutions

We are aware of three existing DL solutions for blocking: DeepER, AutoBlock, and DeepBlock. DeepBlock only marginally uses DL (to improve a non-DL blocking solution) and is described in a 4-page paper with not enough details to implement. DeepER is consistently outperformed by our solutions (e.g., SIF). As a result, we compare our solutions to AutoBlock [79]. AutoBlock learns tuple embeddings using a labeled dataset. For each tuple, it uses LSH to efficiently retrieve top-K nearest neighbors to form the candidate set. We implemented AutoBlock using the details provided in the paper, including identical settings. We used FALCONN [2] to obtain the nearest neighbor using cross-polytope LSH.

Figure 10 shows the results of comparing AutoBlock with Autoencoder and Hybrid, both of which do not require labeled data. We repeated each experiment 5 times and reported the average recall. The curves AB-5, AB-10 and AB-15 shows the results of training AutoBlock with 5%, 10% and 15% of positive matches. Both Autoencoder and Hybrid outperform AutoBlock: they achieve higher recall for a given K. We also evaluated an AutoBlock variant called AB-Hy where we trained AutoBlock using the approximate labeled data generated by Hybrid. This variant's performance is closer to that of Hybrid, but is still below those of our two solutions.

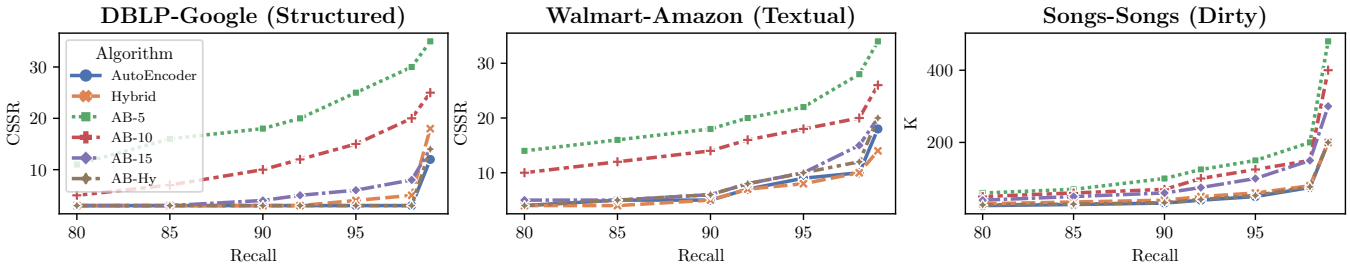

Figure 10: Comparison with AutoBlock variants that use different amounts of training data.

Table 6: Comparing with existing state-of-the-art non-DL solutions.

Datasets	DL			RBB		BSL		TB		Union (DL,RBB)	
	K	|C|	Recall	|C|	Recall	|C|	Recall	|C|	Recall	|C|	Recall
Amazon-Google$_1$	50	68.2k	97.1	16.3k	88.2	123.2k	78.8	489.8k	83.1	77.7k	98.8
Walmart-Amazon$_1$	20	51.1k	92.2	2.1m	92.0	461k	89.1	3.6m	88.2	2.1m	98.9
DBLP-Google$_1$	150	392.4k	98.1	7.3m	96.9	778.2k	92.1	2.9m	96.7	7.6m	99.6
DBLP-ACM$_1$	5	13.1k	99.6	189.7k	95.5	243k	88.5	466k	94.2	198.4k	99.9
Hospital$_1$	150	140k	99.0	90k	99.3	294k	93.2	389k	96.1	209.8k	99.9
Song-Song$_1$	50	50m	95.0	486.1k	94.7	1.2m	92.9	11.2m	92.6	50m	98.7
Amazon-Google$_2$	20	27.3k	70.5	8.4k	60.2	49k	66.3	86.1k	65.3	33.6k	85.0
Walmart-Amazon$_2$	5	12.8k	68.7	7.9m	64.2	86k	58.0	212k	62.6	7.9m	83.0
Abt-Buy	20	21.6k	87.2	28.3k	82.9	43.2k	82.1	289k	81.9	44.6k	95.7
Amazon-Google$_3$	50	68.2k	97.1	320.5k	87.3	730.1k	90.3	3.2m	89.6	360.0k	99.3
Walmart-Amazon$_3$	10	25.5k	88.0	924.3k	87.0	644.1k	86.1	4.4m	84.3	935.9k	97.9
DBLP-Google$_2$	150	392.4k	98.1	47.5m	97.2	1.89m	88.2	8.87m	93.7	47.6m	99.8
DBLP-ACM$_2$	5	13.1k	99.6	991.7k	98.7	2.2m	93.2	3.7m	91.1	1.0m	99.8
Hospital$_2$	10	8.3k	89.0	133.9k	88.4	289k	81.0	569k	79.8	136.8k	98.5
Song-Song$_2$	50	50m	95.0	5.1m	81.9	8.6m	74.4	23.89m	79.7	50m	95.2

5.4 Comparing with Existing Non-DL Solutions

We now compare our solutions with existing state-of-the-art non-DL solutions. From our perspective, these non-DL solutions fall into three categories (a) those that require a domain expert to *manually* select and tune a blocking method, e.g., deciding what kind of hashing to perform on which attributes, (b) unsupervised schema-agnostic approaches, and (c) approaches that learn a blocker from labeled data. As it is challenging to quantitatively compare our solutions with approaches that require manual domain expert effort in (a), we focus on (b) and (c). We choose TB [59] (token blocking) as a representative approach for (b). Based on the recent survey [64], we identify BS [51] (blocking scheme learner) as state-of-the-art approach for (c). They have been shown to achieve good results for diverse datasets [57, 64]. We also compare our solutions to RBB, a state-of-the-art industrial solution that uses labeled data to learn a blocker. In our experiments, we ensure that both BSL and RBB use the same amount of labeled data.

Evaluation Method: It is extremely difficult to vary the parameters of the non-DL solutions (TB, BSL, and RBB) in such a way that can generate meaningful recall-candidate set size ratio curves. As a result, we turn our DL solutions into "point" solutions, so that we can compare them to the non-DL solutions. Specifically, we first select the best DL solution, namely Autoencoder for structured/dirty data and Hybrid for textual data, and then focus on finding a good K value. To do so, we vary K and run DL blocking to generate a candidate set C_K. Then we apply MatchCatcher [45] to C_K to identify the 200 most similar pairs that are not in C_K. If we find more than 10 true matches among these pairs, then we conclude that K is too small to achieve good recall. So we increase K and repeat the previous steps. Otherwise, we stop. Column 2 of Table 6 shows the K value we found for each dataset. Note that the above procedure, perhaps with some minor modifications, can be used in practice to find a good value of K for our DL solutions.

Comparison Results: Table 6 shows the recall and candidate set size for our best DL method (Autoencoder or Hybrid, shown as "DL" in the table), RBB, BSL, and TB, respectively (ignoring the last column "Union (DL, RBB)" for now). The results show that RBB outperforms both the learned-blocker solution BSL and the unsupervised solution TB, in terms of recall and candidate set size. So we focus on comparing our solution with RBB. Our solution achieves higher recall than RBB for all (highlighted in red font) but one dataset. This outperformance is especially stark for textual and dirty datasets, presumably because DL techniques can work better on these datasets. For structured data, our solution outperforms RBB on the first four datasets, with higher recall and smaller candidate set. However, for Hospital$_1$ and Song-Song$_1$, the recalls are comparable, but the size of the candidate set generated by DL is

much larger than the one generated by RBB, especially for Song-Song$_1$. Given these divergent results, it is not clear whether DL will consistently outperform RBB on structured datasets.

5.5 Combining DL and Non-DL Solutions

We found that DL and non-DL solutions learn "concepts" that are highly overlapping but not quite the same. As a result, we conjectured that combining them can significantly increase recall while only minimally to moderately increase the candidate set size. A tuple pair (t_i, t_j) will be passed to the downstream entity matcher if at least one of DL or RBB considers the pair to be a possible match. The last two columns, under "Union (DL,RBB)", of Table 6 shows the candidate set size and recall when we union the output of our DL method and RBB. We see that recall increases significantly (values in blue font) by 0.3-6.7% absolute recall improvement on structured datasets, 8.5-14.5% on textual datasets, and 0.2-9.9% on dirty datasets. The candidate set size increases by up to 49.9%, 57.6%, and 12.3% on structured, textual, and dirty datasets, respectively. But in most cases, the actual size increase is minimal.

5.6 Ablation Analysis

We also conducted additional experiments to validate the various design choices of DeepBlocker. We only summarize the key observations here and refer the reader to [1] for additional details.

1. Sequential vs Joint Training of Hybrid. Currently, Hybrid uses a sequential approach where the aggregator M_1 is trained first followed by that of summarizer M_2. We found that this outperforms the alternative where both M_1 and M_2 are trained jointly. Intuitively, the joint training focuses both M_1 and M_2 on cross-tuple information that produces worst embeddings for in-tuple information.

2. Integrating Attribute Information in Embedding. Our DL models construct a single string by concatenating all the attribute value, which would lead to the loss of information about attribute boundaries. Prior work such as Seq2SeqMatcher [55] found that incorporating attribute information is beneficial for *matching* tuples. By following the same annotation process as [55], for example, tuple a_1 from Figure 1 is encoded as [Name Daniel] [Name Smith] [City LA] [Age 18], we found that the annotation information was not beneficial for blocking where the schema is homogeneous and attributes involved in blocking are aligned. We leave the evaluation of other types of tuple serialization [13, 46] for future work.

3. Making Autoencoder *and* CTT *Sequence-Aware.* We use SIF for aggregating word embeddings to tuple embeddings in both Autoencoder and CTT. We experimented with replacing SIF with a sequence-aware model such as LSTM in Autoencoder and CTT. This provided a minor improvement in textual datasets that can benefit from sequence modeling. While training of Autoencoder using SIF took 3.5 minutes for Abt-Buy, the LSTM variant took as much as 12.4 minutes. Given the negligible improvement in the recall, we believe that it suffices to train using SIF.

4. Evaluating Variants of CTT. By default, CTT takes a pair of tuples t_i and t_j and passes the difference of their tuple embeddings to the classifier. This approach is widely used in prior work including [25, 53]. An alternative is to *concatenate* the embeddings of t_i and t_j and pass it to the classifier. The concatenation approach slightly out-performs CTT due to the higher number of model parameters in the input layer of the classifier. This also requires a larger training data and the consequent higher training cost.

6 RELATED WORK

Section 2 briefly reviews existing non-DL and DL based blocking approaches. An overview of EM can be found in surveys such as [18, 27, 54, 60]. A recent survey [5] specifically focused on the application of DL models for EM. There are also a number of surveys that cover different aspects of blocking [16, 58, 64]. There has been multiple efforts on building scalable EM systems such as Magellan [41], JedAI [62, 65], and CloudMatcher [30].

Recently, a number of approaches have applied transformer based pre-trained language models to EM [13, 46, 47, 66] achieving state-of-the-art results. There has been some effort on reducing the amount of labeled data needed to train DL models using transfer learning [36, 48, 73, 77, 80] and data augmentation [46] with promising results. There has also been some attempt on learning relational embeddings that are targeted towards structured data [10, 11, 14, 31, 42, 74]. These efforts are orthogonal to our work and can be used as a choice for the word embedding module.

MatchCatcher [45] proposed a debugger for blocking that can identify the matches that were ignored by the blocker and use them to improve the blocker's accuracy. There has been extensive work on scaling blocking techniques to single machine and cluster of machines [12, 19, 26, 40]. There has been some effort [8, 20, 51] on learning blocking predicates by leveraging labeled data consisting of matches and non-matches. An empirical comparison of the blocking methods can be found in [6, 57, 72]. Works such as [4, 33] extend the averaging and LSTM based blocking proposed in DeepER.

7 CONCLUSION & FUTURE WORK

In this paper we have significantly advanced the state of the art in applying DL to the blocking step of EM, by defining a large space of DL solutions, developing eight representative solutions in this space, and evaluating these solutions.

Our findings has several implications for researchers and practitioners. First, Autoencoder appears to be a highly promising DL solution. It is relatively simple to implement, fast, yet effective. If the data is highly textual, then perhaps extending Autoencoder with cross-tuple training, i.e., using the Hybrid solution, would improve recall, at the cost of longer runtime. Second, using synthetic labeled data appears to work quite well, and should be investigated further. In the same vein, other types of self supervision should be examined, to see if they can benefit blocking. Third, combining DL and non-DL solutions appears to be a highly promising research direction. In this paper we have examined a simple union to combine the two solutions. One could possibly design better combination methods. Finally, we should investigate other ways to inject domain knowledge into the blocking process, such as the importance of the individual attributes for blocking.

Acknowledgment: We thank the reviewers for invaluable feedback. The last author was supported by UW-Madison UW-2020 grant, NSF grant IIS-1564282, and gifts from Informatica, Google, and American Family Insurance.

REFERENCES

[1] [n.d.]. Tech Report for DeepBlocker. https://www.dropbox.com/s/yirgfecdcyr6aep/DeepBlockerTechReport.pdf?dl=0.

[2] Alexandr Andoni, Piotr Indyk, TMM Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal LSH for angular distance. In *Advances in Neural Information Processing Systems (NIPS 2015)*. 1225–1233.

[3] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. 2017. A Simple but Tough-to-Beat Baseline for Sentence Embeddings. In *ICLR*.

[4] Fabio Azzalini, Songle Jin, Marco Renzi, and Letizia Tanca. 2020. Blocking Techniques for Entity Linkage: A Semantics-Based Approach. *Data Science and Engineering* (2020), 1–19.

[5] Nils Barlaug and Jon Atle Gulla. 2020. Neural networks for entity matching. *arXiv preprint arXiv:2010.11075* (2020).

[6] R Baxter, P Christen, and T Churches. [n.d.]. A Comparison of Fast Blocking Methods for Record Linkage; erschienen in: Proceedings of the Workshop on Data Cleaning, Record Linkage and Object Consolidation at the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining; Washington DC; 2003; o.

[7] Aurélien Bellet, Amaury Habrard, and Marc Sebban. 2013. A survey on metric learning for feature vectors and structured data. *arXiv preprint arXiv:1306.6709* (2013).

[8] Mikhail Bilenko, Beena Kamath, and Raymond J Mooney. 2006. Adaptive blocking: Learning to scale up record linkage. In *Sixth International Conference on Data Mining (ICDM'06)*. IEEE, 87–96.

[9] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2017. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguistics* 5 (2017), 135–146.

[10] Rajesh Bordawekar and Oded Shmueli. 2017. Using word embedding to enable semantic queries in relational databases. In *DEEM Workshop*. ACM, 5.

[11] Rajesh Bordawekar and Oded Shmueli. 2019. Exploiting Latent Information in Relational Databases via Word Embedding and Application to Degrees of Disclosure.. In *CIDR*.

[12] Andrew Borthwick, Stephen Ash, Bin Pang, Shehzad Qureshi, and Timothy Jones. 2020. Scalable Blocking for Very Large Databases. *arXiv preprint arXiv:2008.08285* (2020).

[13] Ursin Brunner and Kurt Stockinger. 2020. Entity matching with transformer architectures-a step forward in data integration. In *EDBT*.

[14] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating embeddings of heterogeneous relational datasets for data integration tasks. In *SIGMOD*. 1335–1349.

[15] Gal Chechik, Varun Sharma, Uri Shalit, and Samy Bengio. 2010. Large scale online learning of image similarity through ranking. (2010).

[16] Peter Christen. 2011. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE transactions on knowledge and data engineering* 24, 9 (2011), 1537–1555.

[17] Peter Christen. 2012. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer.

[18] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. 2020. An overview of end-to-end entity resolution for big data. *ACM Computing Surveys (CSUR)* 53, 6 (2020), 1–42.

[19] Sanjib Das, Paul Suganthan G. C., AnHai Doan, Jeffrey F. Naughton, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, Vijay Raghavendra, and Youngchoon Park. 2017. Falcon: Scaling Up Hands-Off Crowdsourced Entity Matching to Build Cloud Services. In *SIGMOD*.

[20] Anish Das Sarma, Ankur Jain, Ashwin Machanavajjhala, and Philip Bohannon. 2012. An automatic blocking mechanism for large-scale de-duplication tasks. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. 1055–1064.

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 4171–4186.

[22] AnHai Doan, Alon Y. Halevy, and Zachary G. Ives. 2012. *Principles of Data Integration*. Morgan Kaufmann.

[23] Carl Doersch, Abhinav Gupta, and Alexei A Efros. 2015. Unsupervised visual representation learning by context prediction. In *Proceedings of the IEEE international conference on computer vision*. 1422–1430.

[24] Xin Luna Dong, Laure Berti-Equille, and Divesh Srivastava. 2009. Truth discovery and copying detection in a dynamic world. *Proceedings of the VLDB Endowment* 2, 1 (2009), 562–573.

[25] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed representations of tuples for entity resolution. *PVLDB* 11, 11 (2018), 1454–1467.

[26] Vasilis Efthymiou, George Papadakis, George Papastefanatos, Kostas Stefanidis, and Themis Palpanas. 2015. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 411–420.

[27] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate Record Detection: A Survey. *TKDE* 19, 1 (2007).

[28] Chaitanya Gokhale, Sanjib Das, AnHai Doan, Jeffrey F. Naughton, Narasimhan Rampalli, Jude W. Shavlik, and Xiaojin Zhu. 2014. Corleone: hands-off crowdsourcing for entity matching. In *SIGMOD*.

[29] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning*. MIT Press.

[30] Yash Govind, Erik Paulson, Palaniappan Nagarajan, Paul Suganthan G. C., AnHai Doan, Youngchoon Park, Glenn Fung, Devin Conathan, Marshall Carter, and Mingju Sun. 2018. CloudMatcher: A Hands-Off Cloud/Crowd Service for Entity Matching. *Proc. VLDB Endow.* 11, 12 (2018), 2042–2045. https://doi.org/10.14778/3229863.3236255

[31] Michael Günther. 2018. FREDDY: Fast Word Embeddings in Database Systems. In *SIGMOD*. ACM, 1817–1819.

[32] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[33] Delaram Javdani, Hossein Rahmani, Milad Allahgholi, and Fatemeh Karimkhani. 2019. DeepBlock: A Novel Blocking Approach for Entity Resolution using Deep Learning. In *ICWR*. IEEE, 41–44.

[34] Longlong Jing and Yingli Tian. 2020. Self-supervised visual feature learning with deep neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020).

[35] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data* (2019).

[36] Jungo Kasai, Kun Qian, Sairam Gurajada, Yunyao Li, and Lucian Popa. 2019. Low-resource Deep Entity Resolution with Transfer and Active Learning. In *ACL*. 5851–5861.

[37] Yoon Kim, Yacine Jernite, David A. Sontag, and Alexander M. Rush. 2015. Character-Aware Neural Language Models. *CoRR* abs/1508.06615 (2015).

[38] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*.

[39] Diederik P. Kingma and Max Welling. 2019. An Introduction to Variational Autoencoders. *Found. Trends Mach. Learn.* 12, 4 (2019), 307–392.

[40] Lars Kolb, Andreas Thor, and Erhard Rahm. 2011. Parallel sorted neighborhood blocking with MapeReduce. *Datenbanksysteme für Business, Technologie und Web (BTW)* (2011).

[41] Pradap Konda, Sanjib Das, Paul Suganthan GC, AnHai Doan, Adel Ardalan, Jeffrey R Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, et al. 2016. Magellan: Toward building entity matching management systems. *PVLDB* 9, 13 (2016), 1581–1584.

[42] Christos Koutras, Marios Fragkoulis, Asterios Katsifodimos, and Christoph Lofi. 2020. REMA: Graph Embeddings-based Relational Schema Matching. *SEA Data workshop* (2020).

[43] Brian Kulis et al. 2012. Metric learning: A survey. *Foundations and trends in machine learning* 5, 4 (2012), 287–364.

[44] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. 2014. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press.

[45] Han Li, Pradap Konda, Paul Suganthan GC, AnHai Doan, Benjamin Snyder, Youngchoon Park, Ganesh Krishnan, Rohit Deep, and Vijay Raghavendra. 2018. MatchCatcher: A Debugger for Blocking in Entity Matching.. In *EDBT*. 193–204.

[46] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep entity matching with pre-trained language models. *PVLDB* 14, 1 (2020), 50–60.

[47] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, Jin Wang, Wataru Hirota, and Wang-Chiew Tan. 2021. Deep Entity Matching: Challenges and Opportunities. *Journal of Data and Information Quality (JDIQ)* 13, 1 (2021), 1–17.

[48] Michael Loster, Ioannis Koumarelas, and Felix Naumann. 2021. Knowledge Transfer for Entity Resolution with Siamese Neural Networks. *Journal of Data and Information Quality (JDIQ)* 13, 1 (2021), 1–25.

[49] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. 2000. Efficient clustering of high-dimensional data sets with application to reference matching. In *SIGKDD*. 169–178.

[50] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. 2000. Automating the construction of internet portals with machine learning. *Information Retrieval* 3, 2 (2000), 127–163.

[51] Matthew Michelson and Craig A. Knoblock. 2006. Learning Blocking Schemes for Record Linkage. In *AAAI*. 440–445.

[52] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *NeruIPS*. 3111–3119.

[53] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep learning for entity matching: A design space exploration. In *SIGMOD*.

[54] Felix Naumann and Melanie Herschel. 2010. An introduction to duplicate detection. *Synthesis Lectures on Data Management* 2, 1 (2010), 1–87.

[55] Hao Nie, Xianpei Han, Ben He, Le Sun, Bo Chen, Wei Zhang, Suhui Wu, and Hao Kong. 2019. Deep sequence-to-sequence entity matching for heterogeneous entity resolution. In *CIKM*. 629–638.

[56] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748* (2018).

[57] Kevin O'Hare, Anna Jurek, and Cassio de Campos. 2018. A new technique of selecting an optimal blocking method for better record linkage. *Information Systems* 77 (2018), 151–166.

[58] Kevin O'Hare, Anna Jurek-Loughrey, and Cassio de Campos. 2019. A review of unsupervised and semi-supervised blocking methods for record linkage. *Linking and Mining Heterogeneous and Multi-view Data* (2019), 79–105.

[59] George Papadakis, Ekaterini Ioannou, Claudia Niederée, and Peter Fankhauser. 2011. Efficient entity resolution for large heterogeneous information spaces. In *WSDM*. 535–544.

[60] George Papadakis, Ekaterini Ioannou, Emanouil Thanos, and Themis Palpanas. 2021. The Four Generations of Entity Resolution. *Synthesis Lectures on Data Management* 16, 2 (2021), 1–170.

[61] George Papadakis, Georgia Koutrika, Themis Palpanas, and Wolfgang Nejdl. 2013. Meta-blocking: Taking entity resolutionto the next level. *IEEE Transactions on Knowledge and Data Engineering* 26, 8 (2013), 1946–1960.

[62] George Papadakis, George Mandilaras, Luca Gagliardelli, Giovanni Simonini, Emmanouil Thanos, George Giannakopoulos, Sonia Bergamaschi, Themis Palpanas, and Manolis Koubarakis. 2020. Three-dimensional Entity Resolution with JedAI. *Information Systems* 93 (2020), 101565.

[63] George Papadakis, George Papastefanatos, Themis Palpanas, and Manolis Koubarakis. 2016. Scaling entity resolution to large, heterogeneous data with enhanced meta-blocking.. In *EDBT*. 221–232.

[64] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2020. Blocking and filtering techniques for entity resolution: A survey. *ACM Computing Surveys (CSUR)* 53, 2 (2020), 1–42.

[65] George Papadakis, Leonidas Tsekouras, Emmanouil Thanos, Nikiforos Pittaras, Giovanni Simonini, Dimitrios Skoutas, Paul Isaris, George Giannakopoulos, Themis Palpanas, and Manolis Koubarakis. 2020. JedAI3: beyond batch, blocking-based Entity Resolution.. In *EDBT*. 603–606.

[66] Ralph Peeters, Christian Bizer, and Goran Glavaš. 2020. Intermediate training of BERT for product matching. 745, 722 (2020), 2–112.

[67] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *EMNLP*. ACL, 1532–1543.

[68] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *Association for Computational Linguistics* (2019).

[69] Giovanni Simonini, Sonia Bergamaschi, and HV Jagadish. 2016. BLAST: a loosely schema-aware meta-blocking approach for entity resolution. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1173–1184.

[70] Giovanni Simonini, George Papadakis, Themis Palpanas, and Sonia Bergamaschi. 2019. Schema-Agnostic Progressive Entity Resolution. *IEEE Trans. Knowl. Data Eng.* 31, 6 (2019), 1208–1221.

[71] Kostas Stefanidis, Vasilis Efthymiou, Melanie Herschel, and Vassilis Christophides. 2014. Entity resolution in the web of data. In *WWW*. 203–204.

[72] Rebecca C Steorts, Samuel L Ventura, Mauricio Sadinle, and Stephen E Fienberg. 2014. A comparison of blocking methods for record linkage. In *International conference on privacy in statistical databases*. Springer, 253–268.

[73] Saravanan Thirumuruganathan, Shameem A Puthiya Parambath, Mourad Ouzzani, Nan Tang, and Shafiq Joty. 2018. Reuse and adaptation for entity resolution through transfer learning. *arXiv preprint arXiv:1809.11084* (2018).

[74] Saravanan Thirumuruganathan, Nan Tang, Mourad Ouzzani, and AnHai Doan. 2020. Data curation with Deep Learning. *EDBT* (2020).

[75] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NeurIPS*. 5998–6008.

[76] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. 2010. Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion. *J. Mach. Learn. Res.* 11 (2010), 3371–3408.

[77] Renzhi Wu, Sanya Chaba, Saurabh Sawlani, Xu Chu, and Saravanan Thirumuruganathan. 2020. Zeroer: Entity resolution using zero labeled examples. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1149–1164.

[78] Minghe Yu, Guoliang Li, Dong Deng, and Jianhua Feng. 2016. String similarity search and join: a survey. *Frontiers of Computer Science* 10, 3 (01 Jun 2016), 399–417.

[79] Wei Zhang, Hao Wei, Bunyamin Sisman, Xin Luna Dong, Christos Faloutsos, and Davd Page. 2020. AutoBlock: A hands-off blocking framework for entity matching. In *WSDM*. 744–752.

[80] Chen Zhao and Yeye He. 2019. Auto-EM: End-to-end Fuzzy Entity-Matching using Pre-trained Deep Models and Transfer Learning. In *WWW*. 2413–2424.